Patrick GREUSSAY

U.E.R. Informatique-Linguistique
UNIVERSITE de PARIS 8

RT 16-76

Janvier 1976

VLISP : Structure et extensions d'un système LISP pour mini-ordinateurs

Résumé :

Nous décrivons la structure d'un système LISP : VLISP, conçu spécifiquement pour être implémenté sur mini-ordinateurs. Nous en définissons en détail le noyau : interprète et extensions, en utilisant le mode de description par filtrage, ce qui nous permet de ne pas faire dépendre la description d'un mini-ordinateur particulier, tout en permettant d'éviter l'introduction d'une syntaxe abstraite.

Nous introduisons une fonction de contrôle inédite en LISP, dont la puissance nous permet de réaliser avec succès un compromis entre la nécessité d'obtenir un système LISP utilisable, et les restrictions, en particulier en taille mémoire disponible, liées à l'emploi des mini-ordinateurs.

TABLE DES MATIERES

1 INTRODUCTION

Une des caractéristiques principales du langage LISP est la circularité de la définition de ses interprètes (Mc CARTHY 1960a, 1960b, 1962). Toutefois les interprètes publiés du LISP pur, ainsi décrits en LISP pur ont l'inconvénient de ne pas donner des indications précises sur le mode d'implémentation de LISP dans un langage de plus bas niveau *(1)*, et de ne pas refléter l'évolution du langage, tel qu'il est effectivement utilisé dans les études d'Intelligence Artificielle.

Cette évolution s'est traduite par l'extension de la définition de nombreuses fonctions LISP importantes, par le souci d'efficience de l'interprétation, en particulier en ce qui concerne l'accès aux variables, enfin par l'abandon de plusieurs constructions du LISP original, en particulier les expressions LABEL et les objets FUNARG (MOSES 1970, SANDEWALL 1970), qui, quoique d'un grand intérêt théorique (NEWEY 1975, GORDON 1973, 1975), n'étaient pratiquement jamais utilisées en raison de leurs limitations *(2)*.

Ainsi les principaux systèmes LISP actuels : MACLISP (WHITE 1970, MOON 1975), INTERLISP (TEITELMAN 1974, HARALDSON 1975), et LISP 1.6 (QUAM 1973), se révèlent très différents, tant dans leur usage que dans leur définition, du LISP pur original, aussi bien que du LISP 1.5 de 1962. Ces systèmes nécessitent pour leur implémentation des configurations très importantes qui en interdisent pratiquement la transposition sur mini-ordinateurs.

L'implémentation de LISP sur mini-ordinateurs pose alors le problème de trouver un compromis acceptable entre

(1) la puissance du système : la plupart des utilisations de LISP, hautement expérimentales, nécessitent des modifications extrêmement fréquentes des programmes interprétés. Certaines parties pourront demeurer relativement stables et seront naturellement compilées. Reste que la stabilité d'un ensemble de fonctions n'étant pas prévisible à l'avance, exige de l'utilisateur d'employer LISP avec de nombreuses précautions restrictives. Ces restrictions de langage, en particulier l'emploi massif de fonctions de type PROG (LUX 1975) ont pour conséquence un extrême encombrement de la mémoire disponible (LISP 1.6 n'autorise pratiquement que ce style de programmation, de "type-FORTRAN"). La compilation des fonctions LISP est, malgré son élégance apparente, un processus très lourd. De surcroît, il se révèle que sur mini-ordinateurs, les fonctions compilées sont d'un encombrement équivalent, sinon supérieur à celui des fonctions interprétées, et nécessitent de toutes manières de fréquents appels à l'interprète. La situation est exactement inverse dans les systèmes LISP implémentés sur des configurations importantes. Ainsi le très grand nombre de fonctions de faible puissance faciles à compiler des grands systèmes LISP, devra être remplacé sur mini-ordinateurs par un petit nombre de fonctions puissantes.

(1) à l'exception de l'implémentation sur PDP 1 d'une version extrêment réduite de LISP 1.5 (DEUTSCH 1964).

(2) les objets FUNARG ne donnent pas accès à la structure de contrôle, i.e. ne comprennent pas de continuations (cf. §3.3) ; les formes LABEL ne permettent pas la définition *simultanée* de fonctions récursives.

(2) les possibilités d'utilisation effective : les systèmes LISP pour mini-ordinateurs doivent comprendre des fonctions interprétées très puissantes. *Fonctions de contrôle* : expressions conditionnelles étendues, sélection par cas, fonctions de sortie avec restitutions instantanées de contextes, itérations structurées ; *fonctions de données* : filtrage de listes par position ainsi que filtrage par contenu. Ces fonctions permettent ainsi une implémentation aisée et compacte des extensions actuelles de LISP, en particulier de type PLANNER (HEWITT 1975) et CONNIVER (Mc DERMOTT 1974), extensions qui sont devenues à présent indispensables aux études d'Intelligence Artificielle.

Nous nous proposons de décrire le noyau d'un système LISP : VLISP, pour mini-ordinateurs qui réalise ce compromis. Il est clair que nous ne proposons ni un système portable (NORDSTROM 1971, BERTHOD 1976), ni une standardisation de LISP, telle que celle de HEARN (1969), qui de toutes façons ne serait nullement nécessaire. Tout au contraire, nous décrirons en détail l'*implémentation* du système VLISP, reflétant l'état du langage tel qu'il est, à l'heure actuelle, effectivement utilisé.

Cette description sera donnée sous forme de programmes dans une notation elle-même issue des plus récents développements de LISP en fonctions de données : la notation par *filtrage*.

Cette notation nous permettra de décrire la syntaxe concrète des objets LISP créés et manipulés par l'interprète, tout en nous permettant

(1) d'éviter de faire dépendre la description de l'interprète des caractéristiques d'adressage d'ordinateurs particuliers.

(2) de ne pas accéder aux composantes de LISP par des chaînes opaques et fastidieuses de sélecteurs CAR et CDR : notre notation nous permet la référence immédiate à des composantes en position quelconque *(1)*

(3) de ne pas engager prématurément des décisions d'implémentation de trop bas niveau, en particulier en ce qui concerne l'accès aux variables, et le mode d'implémentation de la zone de contrôle de l'interprète.

Cette notation permet de surcroît de rassembler les notions d'affectations, de tests, de spécification de structure d'expressions symboliques LISP, d'accès à des composantes arbitraires de listes, dans une description unifiée.

Avant les descriptions de l'implémentation du système VLISP, nous exposerons notre notation de filtrage, puis nous décrirons les structures de données VLISP et leur organisation en mémoire, ainsi que la spécification en terme de filtrage de nos structures de contrôle sous forme de continuations.

(1) notre notation a donc une puissance descriptive très supérieure à celle proposée par (KOWALSKI 1973), cette dernière ne permettant pas les références directes à des éléments en position arbitraire.

Nous avons effectivement implémenté le système VLISP ici décrit sur plusieurs mini-ordinateurs (en particulier CAE 510, T1600 *(1)*), à l'Université de Paris 8, où plusieurs projets en Intelligence Artificielle y utilisent des extensions décrites dans (GREUSSAY 1975).

(1) VLISP est naturellement transportable sur des ordinateurs plus puissants : une implémentation sur PDP 10 a été réalisée à l'Université de Paris 8 par Jérôme CHAILLOUX (CHAILLOUX 1976).

2 L'OUTIL DE NOTATION : LE FILTRAGE *(1)*

Le filtrage est une expression booléenne que nous noterons

A :: F

dans laquelle *l'argument* A est une S-expression quelconque et F est un *filtre*. Le filtre F est de même une S-expression pouvant également comporter des variables LISP spécialement distinguées que nous nommerons variables de filtre.

Soit $v_1, \ldots, v_n$ les variables de filtre ayant des occurences dans F. Le problème du filtrage est de trouver une suite $s_1, \ldots, s_n$ de S-expressions telles que

$$((\lambda v_1 \ldots v_n \ . \ F) \ s_1 \ldots s_n) = A$$

L'évaluation d'un filtrage est un processus séquentiel et déterministe qui consiste à substituer aux variables de F des éléments constants de A en même position, que nous dirons *filtrés dans* A, de telle façon que F devienne identique à A. Si cela est possible, nous dirons que le filtrage est satisfait, et dans le cas contraire, qu'il échoue.

Exemple : si A = ((1) (2 c d (1)) 2 c d)
et F = (!x (?y d !x) ?y d)

(où x et y seront les variables du filtre F,le préfixe "!" spécifiant x comme variable d'élément, et "?" spécifiant y comme variable de segment)

alors A :: F est satisfait pour la substitution x=(1), y=(2 c)
si encore A = (1 (d 1) d)
alors A :: F est satisfait pour la substitution x=1, y=NIL
mais si A = (1 (d 1) 1 d)
alors il n'existe aucune substitution qui satisfasse A :: F, échec ici du filtrage.

Les variables du filtre seront localement liées à ces éléments constants de A que nous nommerons F-valeurs des variables locales du filtre (en les distinguant des valeurs normales des variables LISP que nous nommerons C-valeurs). En cas de succès du filtrage, la F-valeur des variables du filtre devient la C-valeur des variables LISP correspondantes (par *délocalisation* des variables du filtre). En cas d'échec, la C-valeur des variables du filtre n'est pas modifiée.

2.1 Spécification des filtres

Soit k un atome LISP constant, v une variable, s un segment de A, f un filtre, e une expression LISP évaluable. Les caractères préfixes (cf. §5.4) "!", "?", et "," ont pour rôle de repérer syntaxiquement le type des variables.

(1) le filtrage tel qu'il est ici décrit est une fonction standard de VLISP (cf. §5.9)

Un filtre f pourra être constitué par

(i)	k		un atome LISP. k doit être identique à A.
(ii)	!-		une variable d'élément anonyme : elle filtre A sans contrainte
(iii)	!v	*(1)*	une variable d'élément nommée : elle filtre A sans contrainte et v se trouve localement liée à A.
(iv)	?-		une variable de segment anonyme : elle filtre un segment s le plus court possible de A, de 0 éléments ou plus.
(v)	?v	*(1)*	une variable de segment nommée : elle filtre un segment s le plus court possible de A, et v se trouve localement liée à s.
(vi)	,v		la F-valeur locale de la variable v, si cette liaison locale existe, sinon la C-valeur de v.
(vii)	$!(v\ e_1 \ldots e_n)$		identique à (iii). Toutefois les $e_1 \ldots e_n$ sont des expressions LISP qui imposent des contraintes à la F-valeur de v. Les $e_1 \ldots e_n$ sont évaluées en séquence, et aucune d'entre elles ne doit s'évaluer à NIL.
(viii)	$?(v\ e_1 \ldots e_n)$		identique à (vii) pour les variables de segments.
(ix)	$\{$*ALT* $f_1 \ldots f_n\}$		une suite de filtres $f_1 \ldots f_n$ telle que si A :: f_1 alors succès sinon si A :: f_2 alors succès sinon ... si A :: f_n alors succès sinon échec.
(x)	$(f_1\ f_2 \ldots f_n)$		une suite $f_1 \ldots f_n$ de filtres tels que chacun d'eux doit filtrer un élément ou un segment de A en même position.

En cas d'échec d'un filtre f_i, i>1, une nouvelle tentative est effectuée avec un filtre de segment f_j, i>j, par adjonction d'un élément supplémentaire de A au segment filtré par f_j, et les liaisons locales des variables des filtres $f_{j+1}\ f_{j+2} \ldots f_{i-1}\ f_i$ sont annulées.

Le filtrage échoue totalement s'il y a échec d'un filtre $f_{i>1}$ et qu'il n'existe pas de filtre de segment $f_{j<i}$.

Nous utiliserons également les abréviations suivantes :

$$'x =_{df} (\text{QUOTE } x)$$

$$(x : y) =_{df} (\text{CONS } x\ y)$$

$$(x_1\ x_2 \ldots x_n : y) =_{df} (\text{CONS } x_1\ (\text{CONS } x_2 \ldots (\text{CONS } x_n\ y) \ldots))$$

(1) naturellement, si v, ayant déjà été rencontrée dans le filtre, possède une F-valeur, !v et ?v représentent alors la F-valeur de la variable v.

2.2 Exemples

Si L est une liste non vide

L :: (?- !A1) peut être décrit comme

```
(SETQ A1 (LAST L))
```

(X Y : L) :: !L comme

```
(SETQ L (CONS X (CONS Y L)))
```

L :: (?- (,A1 : !A2) ?-) comme

```
(AND (SETQ X (ASSOC A1 L))
     (SETQ A2 (CDR X)))
```

L :: ((!A1 ?A2) ?L) comme

```
(COND ((LISTP (CAR L))
       (SETQ A1 (CAAR L))
       (SETQ A2 (CDAR L))
       (SETQ L (CDR L))))
```

L :: (?- X ?-) comme

```
(MEMQ 'X L)
```

L :: (?A1 ?A1) filtrant un argument de type (*segment même-segment*) et affectant si c'est le cas le segment de liste à A1 peut être décrit comme

```
(COND
   (L (AND
        (EVENP (LENGTH L))
        (EQUAL (SETQ Y
                  (NTH (SETQ X (ADD1 (QUO (LENGTH L) 2)))
                       L))
               (REVERSE (NTH X (REVERSE L))))
        (SETQ A1 Y)))
   ((SETQ A1 NIL)))
```

3 LES OBJETS DECRITS : atomes, liste de contrôle, continuations

3.1 ATOMES

Bien qu'un atome LISP soit en principe un objet indécomposable par des moyens chimiques (CAR et CDR), il convient, s'il s'agit de décrire son implémentation, de lui imposer une certaine structure.

Nous distinguerons deux types d'atomes

- les *nombres* : un nombre n sera représenté par une S-expression (*nombre* : n)
- les *atomes non-numériques* : un tel atome x sera représenté par la donnée de quatre champs

  <x.CVAL, x.PLIST, x.CODE, x.NOM>

  tels que

  x.CVAL : valeur LISP de l'atome x considéré comme variable

  x.PLIST : Γ-liste de l'atome x

  x.CODE : continuation associée à x dans l'interprète si x est une fonction standard

  x.NOM : nom imprimable de l'atome x.

3.2 LISTE DE CONTROLE

Il s'agit d'une liste quelconque, pouvant donc être argument d'un filtrage. Elle pourra comprendre

- des atomes
- des listes LISP
- des continuations de l'interprète

Elle sera le plus souvent gérée comme une pile. Notre méthode de description par filtrage nous permettra, au niveau de l'implémentation, de rendre cette pile *visible*.

3.3 CONTINUATIONS

Une continuation c est une suite ordonnée de filtrages *(1)*

$$f_1 ; \ldots ; f_n$$

(1) la technique des continuations (FISHER 1972, STRACHEY 1973) consiste à ajouter à une fonction $f(x_1,\ldots,x_n)$ un argument fonctionnel supplémentaire y, la continuation, et à transformer f en une nouvelle fonction

$$f'(x_1,\ldots,x_n,y) = y(f(x_1,\ldots,x_n))$$

telle que f' ne retourne pas une valeur, mais retourne une combinaison qui est l'application de la continuation y à la valeur calculée par f.
Notre interprétation des continuations est fondée sur la possibilité de *défonctionaliser* une continuation en une liste finie d'instructions (REYNOLDS 1972).

Nous distinguerons des continuations *nommées*, par exemple

EVAL = c

et des continuations *anonymes*, par exemple

alors c
sinon c

Nous utiliserons pour décrire l'interprète VLISP deux variables de continuation : C qui désigne la *continuation présente* et R qui désigne une *continuation de retour*. Nous nommerons *filtrage actif* le premier élément de la continuation présente.

Nous utiliserons les structures de contrôle suivantes :

soit C1 et C2 des continuations et P une liste de contrôle

((c2 : P) c1) :: (!P !C) abrégé en *rec* c1 ; c2

qui est interprété naturellement comme un appel récursif, la continuation c2 est empilée dans la liste de contrôle, c1 devient la continuation présente et son premier élément devient le filtrage actif.

P :: (!C ?P) abrégé en *derec*

retour d'appel récursif.

(c1 c2) :: (!C !R) abrégé en *app* c1 ; c2

qui est interprété naturellement comme un appel non nécessairement récursif, et non-nécessairement suivi de retour.

R :: !C abrégé en *ret*

A tout filtrage sera associé une continuation exceptionnelle qui deviendra la continuation présente en cas d'échec du filtrage.
Nous distinguons ainsi les *expressions conditionnelles*

si A :: F *alors* c1 *fsi* c2
et *si* A :: F *alors* c1 *sinon* c2 *fsi*

dans lesquelles la continuation exceptionnelle est c2,
et les *expressions inconditionnelles*

; A :: F ;

dans lesquelles la continuation exceptionnelle est dépendante d'une implémentation particulière, l'interprétation usuelle étant la communication à l'utilisateur du système VLISP d'un diagnostic d'erreur, suivie d'une activation de la boucle-racine-LISP.

Par commodité, nous utiliserons également des abréviations telles que

tantque f *faire* c *ftan*
pour E = *si* f *alors* c ; *app* E *fsi*

4 L'ORGANISATION EN MEMOIRE DES OBJETS DECRITS

4.1 Structure de la mémoire

La mémoire de VLISP se divise en cinq zones

(1) la zone de l'interprète, comprenant les tables syntaxiques, les tampons et programmes d'entrée-sortie, et les continuations de l'interprète proprement dit.

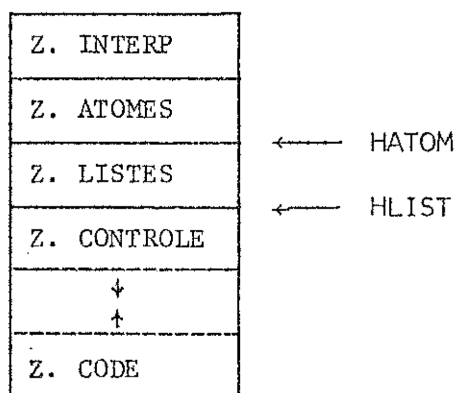
(2) la zone des atomes non-numériques, limitée supérieurement par le pointeur HATOM.

(3) la zone des listes et des nombres, limitée supérieurement par le pointeur HLIST.

(4) la zone de contrôle, cette zone étant le plus souvent gérée comme une pile, limitée supérieurement par

(5) la zone des programmes compilés et assemblés des utilisateurs.

En VLISP T1600 (figure 1) ces zones sont consécutives en mémoire.



- figure 1 -

Un élément de liste ou un nombre occuperont deux mots de 16 bits consécutifs de la zone des listes. Le type, atome ou liste d'un objet est donc déterminé par son adresse d'implantation.

Dans le système VLISP T1600, les dimensions des zones atomes, listes, et contrôle sont modifiables dynamiquement sur une intervention de l'utilisateur, dans la limite des ressources disponibles.

4.2 Organisation de la zone des atomes

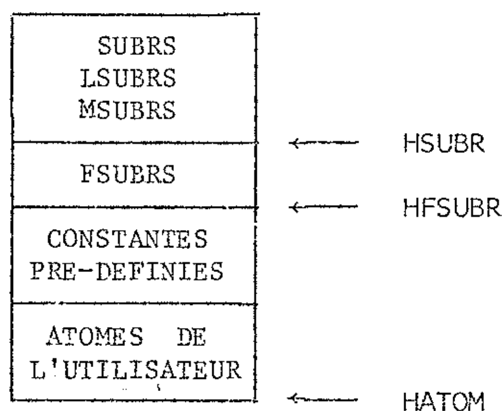
Cette zone, en VLISP, se répartit en quatre sous-zones

(1) les atomes pré-définis qui sont les noms de fonctions standard de type SUBR, LSUBR, MSUBR. Cette sous-zone est limitée supérieurement par le pointeur HSUBR.

(2) les atomes pré-définis qui sont les noms de fonctions standard de type FSUBR, sous-zone limitée supérieurement par le pointeur HSUBR.

(3) les atomes pré-définis qui sont des constantes standard : NIL, QUOTE,T, LAMBDA, EXPR, FEXPR.

(4) les atomes qui sont définis par l'utilisateur : constantes, variables et noms de fonctions



- figure 2 -

En VLISP T1600 ces sous-zones sont consécutives en mémoire. Le type des fonctions standard est défini par l'adresse d'implantation de leur atome-nom, et non pas par la présence ou l'absence d'un indicateur.

4.3 Implementation des atomes

Un atome sera une suite de mots consécutifs en mémoire comprenant

(1) deux mots, CVAL et PLIST, pouvant contenir l'adresse d'un atome ou d'une liste.

(2) un mot : CODE pouvant contenir l'adresse d'une continuation de l'interprète.

(3) une zone : NOM pouvant recevoir une chaîne de caractères qui rendra imprimable le nom de l'atome.

En VLISP T1600, les zones CVAL et CODE sont confondues (figure 3) *(1)*

CODE	PLIST
N O M	

Atome de fonction standard

CVAL	PLIST
N O M	

Constante standard
ou
atome de l'utilisateur

- figure 3 -

Un atome x, pré-défini ou non est une constante si

x.CVAL = x

Les zones CVAL et PLIST sont organisées en doublet et sont donc accessibles de la même façon qu'un élément de liste

(CAR x) = x.CVAL
(CDR x) = x.PLIST

Les fonctions CAR et CDR sont donc définies avec des arguments atomiques. Le doublet (CVAL.PLIST) d'un atome est dès lors modifiable par les fonctions RPLACA, SET et SETQ pour la zone CVAL, et par la fonction RPLACD pour la zone PLIST.

La zone PLIST des atomes standard pourra contenir des informations intéressantes sur l'état de l'interprète, ainsi l'évaluation de

(CDR 'REM)

livre le nombre de doublets disponibles depuis le dernier garbage-collecting.

(1) Ces zones sont au contraire distinctes en VLISP PDP 10, et des fonctions standard de modification de la zone CODE permettent une certaine compatibilité avec d'autres systèmes LISP, où les mêmes fonctions peuvent avoir des noms différents, notamment MACLISP et INTERLISP.

5 DESCRIPTION DE L'INTERPRETE

5.1 Le noyau

La boucle-racine-LISP est la continuation exceptionnelle de tout filtrage inconditionnel. Elle devient la continuation présente en cas d'échec du filtrage.

Le mode EVALQUOTE étant tout à la fois une incommodité et une source de confusion, l'interprète VLISP fonctionne naturellement en mode EVAL.

```
        BOUCLE-RACINE-LISP =
                    repeter rec READ ;
                            rec EVAL ;
                            rec PRINT
                    frep
```

Le rôle principal d'EVAL et APPLY est le lancement des fonctions standard par l'expression : *app* F.CODE. APPLY et EVAL peuvent également lancer des SUBRS et LSUBRS (cf. §5.4), la liste des arguments évalués est dans A4, les premiers, seconds et troisièmes éléments de cette liste sont distribués dans A1, A2, A3, pourvoyant ainsi aux fonctions standard à 1, 2 ou 3 arguments. EVAL seul peut lancer les FSUBRS, avec dans A1 la liste des arguments non-évalués.

5.1.1 eval

```
    EVAL =
            si     A1≤HATOM alors A1.CVAL :: !A1
            sinsi A1 :: {*ALT* (nombre ?-) (QUOTE !A1)} alors
            sinon A1 :: (!F ?A1);
    EVAL2 =
                  si F≤HATOM alors
                     si    F.PLIST :: (?- EXPR !F ?-) alors
                     sinsi F.PLIST :: (?- FEXPR !F ?-) alors
                           (F (A1:NIL)) :: (!A1 !A4);
                           app APPLY
                     sinsi F≤HSUBR alors (marq :F) :: !F
                     sinsi F≤HFSUBR alors app F.CODE
                     sinon (F F.CVAL) :: (!VARESC !F);
                           app EVAL2
                     fsi
                  fsi
                  (F : P) :: !P; rec EVLIS; (A1 P) :: (!A4 (!A1 ?P));
                  si A1 :: (marq ?F) alors
                     A4 :: (!A1 !A2 !A3); app F.CODE
                  sinon app APPLY
                  fsi
            fsi
            derec
```

Lorsque le premier élément de la forme à évaluer est un atome :

(1) si c'est le nom d'une EXPR : les arguments sont évalués et main est passée à APPLY

(2) si c'est le nom d'une FEXPR : APPLY est alors directement invoquée

(3) si c'est le nom d'une SUBR : les arguments sont évalués et celle-ci est lancée

(4) si c'est le nom d'une FSUBR : lancement direct de celle-ci

(5) sinon, faute de disposer de l'information suffisante pour savoir si les arguments doivent être évalués, la CVAL de l'atome est extraite, et la continuation EVAL2 est relancée.

Lorsque le premier élément de la forme à évaluer n'est pas atomique, EVAL suppose par défaut qu'il s'agira d'une EXPR ou d'une SUBR, les arguments sont évalués, et main est passée à APPLY.

5.1.2 apply

```
APPLY =
        repeter
                si A1≤HATOM alors
                   si    A1.PLIST :: (?- EXPR !A1 ?-) alors
                   sinsi A1≤HSUBR alors
                         (A1 : A4) :: (!F !A1 !A2 !A3); app F.CODE
                   sinon A1.CVAL :: !A1
                   fsi
                sinsi (A4 : A1) :: (!Y LAMBDA !X ?A1) alors
                      app LIER; rec PROGN; app DELIER; derec
                sinon
                      (A4 : P) :: !P; rec EVAL; P :: (!A4 ?P)
                fsi
        frep
```

APPLY doit trouver dans A4 une liste d'arguments non-évalués, et dans A1 une expression qui directement ou indirectement devra être une lambda-expression, ou le nom d'une SUBR.

Si APPLY trouve dans A1 le nom d'une EXPR, l'expression associée à cet indicateur sur la P-liste est extraite. Nous aurons les possibilités

(1) (EXPR !λ-expression) : les variables locales sont alors liées, et le corps de la λ-expression est évalué

(2) (EXPR !nom-de-SUBR) : ce qui est une façon de rebaptiser les fonctions standard

(3) (EXPR !variable) : la CVAL de la variable est extraite et APPLY est relancée

Si APPLY trouve dans A1 le nom d'une SUBR, celle-ci est lancée. Sinon l'évaluation du contenu de A1 est demandée, après quoi APPLY est relancée.

Nous examinerons en détail au §5.3 les continuations LIER et DELIER, responsables de la sauvegarde et de la restitution des valeurs des variables.

5.1.3 evlis et progn

```
EVLIS =
        si     A1 :: NIL alors
        sinon (NIL : NIL) :: !A2; (A2 : P) :: !P;
              tantque (A1 A2 A1 : P) :: ((!A1 ?-) ?P)  faire
                      rec EVAL;
                      ((A1 : NIL) : P) :: (!A2 !A3 (!- ?A1) ?P);
                      A2 :: !A3.CDR
              ftan
              P :: ((!- ?A1) ?P)
        fsi
        derec

PROGN =
        A1 :: !A2;
        tantque (A2 A2 : P) :: ((!A1 ?-) ?P) faire
                rec EVAL; P :: ((!- ?A2) ?P)
        ftan
        derec
```

PROGN et EVLIS sont, en VLISP, les opérateurs de *séquentialité*.

EVLIS reçoit en A1 une liste d'arguments non-évalués. Elle doit retourner à son appelant une liste de ces arguments évalués, liste retournée en A1. On notera que cette évaluation est strictement séquentielle.

PROGN reçoit de même en A1 une liste de formes à évaluer. La valeur de ces formes est ignorée à l'exception de celle de la dernière, qui est retournée en A1.

PROGN pourrait ainsi se définir en VLISP

```
(DE PROGN (L) (COND
    ((CDR L) (EVAL (NEXTL L)) (PROGN L))
    ((EVAL (CAR L)))))
```

5.1.4 Un exemple d'évaluation

Suivons notre interprète dans l'évaluation d'un exemple.

Après l'évaluation de

```
(SETQ B 'CAR) (SETQ A '(CONS))
```

nous aurons

```
B.CVAL = CAR
A.CVAL = (CONS)
```

Est alors livrée à EVAL la forme

```
((B A) 'V 'W)
```

Nous suivrons le destin de cette forme en utilisant la notation

$$x\{v_1 = e_1, \ldots, v_n = e_n\}$$

qui donne les valeurs $e_1, \ldots, e_n$ des variables $v_1, \ldots, v_n$ de l'interprète dans la continuation x.

```
eval  {A1 = ((B A) 'V 'W)}
eval  {F  = (B A), A1 = ('V 'W)}
eval2 {A1 = (B A), A4 = (V W)}
apply {A1 = (B A), A4 = (V W)}
apply {eval  {A1 = (B A)},        A4 = (V W)}
apply {eval  {F  = B,    A1 = (A)}, A4 = (V W)}
apply {eval2 {F  = CAR, A1 = (A)}, A4 = (V W)}
apply {eval2 {F  = (marq : CAR), A1 = (A)}, A4 = (V W)}
apply {eval2 {A1 = (marq : CAR), A4 = ((CONS))}, A4 = (V W)}
apply {eval2 {F  = CAR, A1 = (CONS), A2 = A3 = NIL, A4 = ((CONS))}, A4 = (V W)}
apply {A1 = CONS, A4 = (V W)}
apply {F  = CONS, A1 = V, A2 = W, A3 = NIL, A4 = (V W)}
 ...  {A1 = (V . W)} ...
```

On notera qu'EVAL considère les définitions de fonction de l'utilisateur comme prioritaires sur la définition standard. Ainsi l'utilisateur a tout loisir de redéfinir une fonction standard. Voici en exemple une redéfinition de SETQ

```
(DF SETQ (1L)
    (SET (PRIN1 (CAR 1L)) (PROGN
       (PRIN1 '=) (PRINT (EVAL (CADR 1L))))))
```

Chaque évaluation d'une affectation de variable provoquera l'impression du nom de cette variable ainsi que de la valeur de la variable ainsi affectée

(SETQ A '(LE DOME))

provoquera l'impression de

" A = (LE DOME) "

et A.CVAL = (LE DOME)

On notera également qu'EVAL est une fonction disponible à l'utilisateur, fonction standard de type SUBR (cf. §5.4). Ainsi on aura :

```
(EVAL  (ADD1 1)) → 2
(EVAL '(ADD1 1)) → 2
(EVAL  (READ)) (ADD1 1) → 2
```

5.2 Lambda-expressions étendues

En VLISP, une lambda-expression est de la forme

(λ liste-de-variables e1 e2 ... en)

dans laquelle les e1, e2, ... , en sont évalués en séquence par la continuation PROGN qui retourne la valeur de en . Ceci permet d'éviter de réévaluer des sous-expressions communes, et introduit une forme de séquentialité plus efficiente que celle permise par les formes du type PROG.

On notera l'absence des formes LABEL qui, en LISP 1.5 autorisaient des définitions provisoires de fonctions, ainsi que la disparition des formes du type

(FUNARG fonction a-liste)

Ceci est une conséquence du mode de liaison des variables en VLISP où la stratégie dite d'élimination (FISCHER 1972) a été délibérément choisie. Cette caractéristique peut être partiellement remplacée par la fonction, de type SUBR

(EVALA e a-liste)

dont voici l'implémentation

$EVALA_{subr}$ = *si* A2 :: (?- (,A1 ?A1) ?-) *alors derec*
sinon app EVAL
fsi

et qui donne pour l'évaluation de variables, priorité à la liaison de celles-ci si elle existe, sur une A-liste de l'utilisateur, et dans le cas contraire invoque normalement EVAL.

5.3 Liaisons des variables

VLISP se distingue *très fortement* de LISP 1.5 en mettant en jeu le principe des *variables fluides* i.e. la portée des variables libres étant définie dynamiquement.

En LISP 1.5. les variables étaient localement liées sur une A-liste, représentable fonctionnellement, pour une A-liste de longueur n, par l'expression conditionnelle

(λx . *si* x = var1 *alors* val1 *sinsi* x = var2 *alors* val2 ...
sinsi x = varn *alors* valn
sinon erreur-variable-indéfinie
fsi)

Le principe de la A-liste rendait en LISP 1.5 très facile la liaison locale et la restitution des valeurs des variables, mais était très inefficiente tant en consultation qu'en modification *(1)* et obligeait de surcroît à distinguer deux types de variables

(1) les variables globales dites SPECIALES, affectables par RPLACA ou CSETQ

(2) les variables de la A-liste, affectables par SET et SETQ

En VLISP il n'y a pas de distinction entre variables locales et globales. Toutes les variables sont globales et leurs valeurs sont consultables et modifiables immédiatement dans la zone CVAL des atomes. La fonction d'affectation SETQ peut donc s'appliquer à toutes les variables. Ainsi les variables libres sont liées à la racine de LISP qui constitue un environnement implicite de liaison pour toutes les variables.

La C-valeur des variables est, à la lecture de leur atome-nom, initialisée avec avec une valeur spéciale : *INDEFINI* ce qui permet de repérer les tentatives de consultation de variables non encore affectées.

(1) de surcroît, les A-listes, en encombrant la zone des listes, rendaient d'autant plus fréquentes les garbage-collections.

C'est APPLY qui, lors de l'évaluation d'une forme de type

$$((\lambda\ (v_1\ \ldots\ v_n)\ \text{corps})\ e_1\ \ldots\ e_n)$$
ou
$$((\lambda\ v\ \text{corps})\ e_1\ \ldots\ e_n)$$

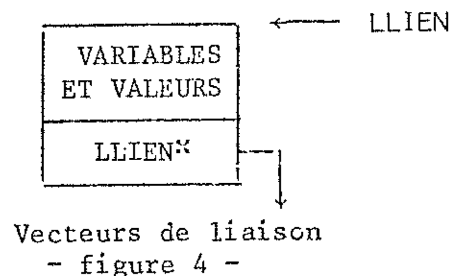
assure la gestion des liaisons et déliaisons des variables locales à la lambda-expression, en appelant les continuations LIER et DELIER

```
LIER =
        ((marq : LLIEN) :P) :: !P;
        si X≤HATOM alors
           si     X :: NIL alors
           sinon (Y X X.CVAL : P) :: (!X.CVAL ?P)
           fsi
        sinon tantque X :: (!X1 ?X) faire
                      (Y X1 X1.CVAL : P) :: ((!X1.CVAL ?Y) ?P)
              ftan
        fsi
        P :: !LLIEN; ret

DELIER =
        repeter P :: (!X ?P);
                si     X :: (marq ?LLIEN) alors ret
                sinon P :: (!X.CVAL ?P)
                fsi
        frep
```

LIER attend dans X une variable (cas des LEXPR) ou une liste de variables. Y devra contenir une liste de valeurs à lier localement à ces ou cette variables.

En VLISP T1600, la gestion des variables est assurée au moyen d'une suite chaînée de *vecteurs de liaisons* (figure 4), elle même sous-liste de la liste de contrôle.



Vecteurs de liaison
- figure 4 -

Confrontons sur un exemple les méthodes de liaison de LISP 1.5 et VLISP. Supposons que la continuation APPLY devient la continuation présente, avec

A1 = (λ (U V W) ?corps)
A4 = (8 (DOME) B)

En LISP 1.5 nous aurons :

(1) avant liaison, l'état de la A-liste, par exemple

((U.FEU) (V.3) (W.-12) ...)

(2) après liaison

((U.8) (V.DOME) (W.B) (U.FEU) (V.3) (W.-12) ...)

En VLISP T1600 nous aurons

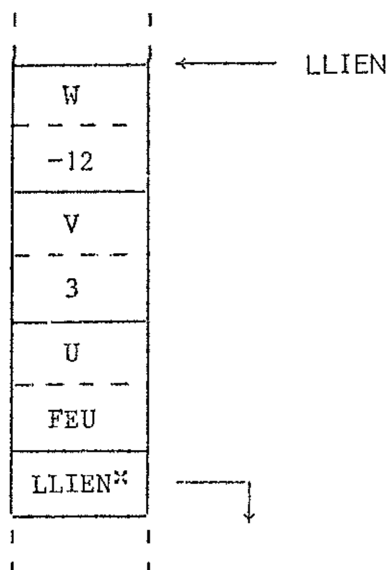
(1) avant liaison :

U.CVAL = FEU   V.CVAL = 3   W.CVAL = -12

(2) après liaison :

U.CVAL = 8   V.CVAL = (DOME)   W.CVAL = B

et un nouveau vecteur de liaison sera formé dans la liste de contrôle (figure 5)



- figure 5 -

L'affectation de *toutes* les variables sera possible en VLISP grâce aux fonctions SET, SETQ et RPLACA.
Voici l'implémentation de SETQ

```
SETQ fsubr =
          (A1 A1 : P) :: ((!- !A1) ?P); rec EVAL;
          P :: ((!A2 ?-) ?P); A1 :: !A2.CVAL;
          derec
```

En outre, VLISP comprend une fonction très particulière

(NEXTL !variable)

qui retourne le CAR de la valeur de la variable, et réaffecte implicitement la variable avec le CDR de sa valeur, ce qui permet l'utilisation d'une forme limitée de *stream* (LANDIN 1965).

```
NEXTL fsubr =
          A1 :: (!A2); A2.CVAL :: (!A1 ?A2.CVAL); derec
```

Soit	L.CVAL = (a b c)
l'évaluation de	(SETQ X (NEXTL L))
donnera	X.CVAL = a
et	L.CVAL = (b c)

5.4 Types des fonctions en VLISP

En VLISP, les fonctions, qu'elles soient standard ou définies par l'utilisateur sont distinguées par

- . le traitement (évaluation et distribution) des arguments des appels
- . l'appel explicite ou implicite (macros)

On distingue en VLISP 8 types de fonctions

(1) SUBRS : fonctions standard de nombre fixe d'arguments évalués et distribués
(2) EXPRS : fonctions de l'utilisateur identiques à (1)
(3) FSUBRS : fonctions standard de nombre quelconque d'arguments non-évalués et rassemblés en une liste unique
(4) FEXPRS : fonctions de l'utilisateur de nombre quelconque d'arguments non-évalués et rassemblés en une liste unique liée au premier argument
(5) LSUBRS : fonctions standard, de nombre quelconque d'arguments évalués rassemblés en une liste unique
(6) LEXPRS : fonctions de l'utilisateur, identiques à (5)
(7) MSUBRS : macros-standard. La détection dans le flot de lecture du caractère qui est le nom de la MSUBR provoque le lancement de la continuation de l'interprète correspondante, interrompant ainsi provisoirement la lecture
(8) MEXPRS : macros de l'utilisateur, identiques à (7)

Ainsi l'abréviation 'x pour (QUOTE x) est implémentée par une macro de type MSUBR qui, si elle n'était pas standard serait ainsi définie

```
(MCHAR ' (λ () (LIST QUOTE (READ))))
```

En voici un autre exemple. Si on a défini

```
(MCHAR = (λ () (REVERSE (READ)))
```

alors l'évaluation consécutive à la lecture de

```
'(a b =(c d e) f g)
```

retournera en valeur

```
(a b (e d c) f g)
```

La possibilité de définir des macro-caractères est nécessaire pour l'implémentation de dérivés de LISP tels que CONNIVER (Mc DERMOTT 1974), qui imposent des types aux variables des filtres *(1)*

Les fonctions de l'utilisateur, même définies avec n arguments peuvent en VLISP être appelées avec un nombre k d'arguments différents

(1) si k>n les arguments supplémentaires seront ou non évalués, selon le type de la fonction, puis ignorés
(2) si k<n les arguments de la définition seront initialisés commodément à la valeur NIL.

Cette non-obligation d'identité du nombre d'arguments actuels et formels, ainsi que la définition précise par l'implémentation de EVLIS de la séquentialité de l'évaluation des arguments d'une fonction EXPR, LEXPR, SUBR ou LSUBR, permettent d'utiles effets de bord.

(1) voir (BAUMGART 1972) pour une illustration de l'opacité de lecture des programmes MICRO-PLANNER, inhérente à la non-implémentation en LISP 1.6 des macros-caractères.

Ainsi (SET 'X Y (SETQ Y X))
échange les C-valeurs des variables X et Y.
Ainsi encore si on a évalué

```
(SETQ X '(NULL))
(CAR X) → NULL
((CAR X)) → T
((CAR X) ((CAR X))) → NIL
```

5.5 Les P-listes

Le type des fonctions standard est déterminé par l'adresse d'implémentation de leur atome-nom. Celui des fonctions de l'utilisateur est déterminé par consultation d'un indicateur sur leur P-liste.

Les fonctions de consultation et de modification de P-liste sont, en VLISP, GET et PUT

```
GET_subr =
        si    A1≤HATOM alors A1.PLIST :: !A1 fsi
        si    A1 :: (?- ,A2 !A1 ?-) alors
        sinon NIL :: !A1
        fsi
        derec

PUT_subr =
        si A1≤HATOM alors
           A1.PLIST :: !A4;
           si A4 :: () alors
              (A3 A2) :: !A4.PLIST; derec
           fsi
        sinon A1 :: !A4
        fsi
        jusqua A4 :: (,A3 ?A4) faire
                A4 :: (!- ?A4);
                si A4 :: (!-) alors
                   (A3 A2) :: !A4.CDR; derec
                sinon A4 :: (!- ?A4)
                fsi
        fjusq
        A2 :: !A4.CAR; derec
```

Les fonctions standard DE et DF réalisent la définition de fonctions de type EXPR-LEXPR et FEXPR en créant la lambda expression attachée à l'indicateur de type sur la P-liste du nom de la fonction. Voici par exemple l'implémentation de DE

```
DE_fsubr=
        A1 :: (!A1 ?A2);
        (EXPR LAMBDA : A2) :: (!A3 ?A2); app PUT
```

Toutefois une P-liste, en VLISP peut ne pas être liée à un atome. GET et PUT sont donc définies sur des P-listes autonomes.

Ainsi (GET '(A 1 B 2) 'B) = 2
et (PUT '(A 1 B 2) 'C 3) = (A 1 B 2 C 3)

Cette implémentation des P-listes rend aisée la définition de MEMO-FONCTIONS (MICHIE 1967), objets à mi-chemin entre procédures et tables consultables.

Voici par exemple la définition sous forme de mémo-fonction de la fonction FIBONACCI en VLISP, telle qu'aucun appel de la fonction ne sera calculé plus d'une fois, pour toute valeur de l'argument.

```
(DE FIB (N)
        (OR (GET 'FIB N)
            ((λ (X) (PUT 'FIB X N)
                    X)
            (COND
               ((LT N 2) 1)
               ((PLUS (FIB (SUB1 N))
                      (FIB (DIFFER N 2))))))))
```

5.6 Expressions conditionnelles, sélection par cas et itérations

Le COND de LISP 1.5 a été étendu, sous la forme *(1)*

(COND c_1 c_2 ... c_n)

dans laquelle chaque clause c_i est de la forme

(e_1 e_2 ... e_m)

le premier élément e_1 de la clause c_1 est évalué, et, si la valeur retournée est non-NIL, les e_2 ... e_m de la classe sont évalués en séquence, et la valeur de e_m est retournée comme valeur du COND. Si toutefois l'évaluation de e_1 livre NIL, la clause c_2 est traitée de la même façon. Si aucune des clauses n'est retenue, la valeur du COND est NIL.Une clause c_i peut être réduite à l'unique élément (e_1), et si son évaluation livre non-NIL, cette valeur est retenue comme valeur du COND.

Voici l'implémentation de COND

```
COND_fsubr =
        tantque (A1 A1 : P) :: (((!A1 ?-) ?-) ?P) faire
                rec    EVAL;P :: (!A2 ?P);
                si     A1 :: NIL alors A2 :: (!- ?A1)
                sinon A2 :: ((!- ?A2) ?-);
                       si     A2 :: NIL alors derec
                       sinon A2 :: !A1; app PROGN
                       fsi
                fsi
        ftan
        derec
```

En VLISP, les fonctions AND et OR sont des structures de contrôle et retournent, dans le cas du AND, NIL ou la valeur du dernier argument, et dans le cas du OR, NIL ou la valeur du premier argument tel que son évaluation retourne non-NIL.

Voici l'implémentation du AND

```
AND_fsubr =
        repeter (A1 A1 : P) :: ((!A1 ?-) ?P);
                rec    EVAL; P :: ((!- ?A2) ?P);
                si     A2 :: NIL alors derec
                sinsi A1 :: NIL alors derec
                sinon A2 :: !A1
                fsi
        frep
```

(1) cette extension du COND est due à D.G. BOBROW (BOBROW 1969).

La fonction de sélection par cas SELECTQ a été étendue, sous la forme

(SELECTQ e c_1 c_2 ... c_n)

où e est une expression LISP évaluable et chaque clause c_1, ..., c_{n-1} est de la forme

(atome suite-des-expressions-à-évaluer)

L'expression e est évaluée, puis comparée à l'atome de chacune des clauses c_1, ..., c_{n-1}. S'il y a identité, la valeur du SELECTQ est celle de la suite-des-expressions correspondante. S'il n'y a identité pour aucune, la dernière clause c_n est évaluée par PROGN.

```
SELECTQfsubr =
          (A1 A1 : P) :: ((!A1 ?-) ?P);
          rec EVAL; P :: ((!- ?A2) ?P);
          tantque A2 :: (!A3 !- ?-) faire
                  si A3 :: (,A1 ?A1) alors
                     app PROGN
                  fsi
                  A2 :: (!- ?A2)
          ftan
          A2 :: (!A1); app PROGN
```

Plusieurs formes d'itérations sont implémentées, en particulier la fonction

(WHILE e_1 ... e_n)

qui évalue séquentiellement les e_2 ... e_n tant que l'évaluation de e_1 retourne la valeur non-NIL.

```
WHILEfsubr =
        (A1 : P) :: !P;
        repeter P :: ((!A1 ?-); rec EVAL;
                si    A1 :: NIL alors P :: (!- ?P);
                sinon P :: ((!- ?A1) ?-); rec PROGN
                fsi
        frep
```

Les fonctions de type MAPC fournissent une autre forme d'itération dans laquelle une fonction, second argument du MAPC est appliquée à chaque élément d'une liste, premier argument du MAPC

(MAPC !e !fonction)

la fonction argument peut être une fonction standard, une fonction définie par l'utilisateur, ou encore une lambda-expression.

```
MAPCsubr =
       (A2 : P) :: !P;
       tantque (A1 A1 : P) :: ((!A4 ?-) ?P) faire
               ((A4 : NIL) P) :: (!A4 !- !A1 ?-);
               rec APPLY; P :: ((!- ?A1) ?P)
       ftan
       P :: (!- ?P); derec
```

5.7 Le problème des PROGS

PROG et ses fonctions associées, RETURN et GO, survivances en LISP de la "programmation de type FORTRAN" (Mc CARTHY 1960a) sont implémentés en VLISP essentiellement pour conserver la compatibilité avec d'autres versions de LISP. C'est probablement l'emploi du PROG qui a été à la source des critiques les plus justifiées adressées à l'usage de LISP sous interprète

(1) inefficience remarquable due au balayage du corps de fonction à la recherche des étiquettes, balayage renouvellé à chaque appel du PROG *(1)*
(2) encombrement de la zone des listes, en LISP 1.5, par les tables de symboles ainsi constituées, encombrement également redoublé à chaque appel. Ce défaut reste présent dans une moindre mesure en LISP 1.6 (QUAM 1972) lorsque ces tables de symboles sont implantées dans la pile de contrôle
(3) inefficience des branchements de type (GO etiquette), inévitablement précédés d'une recherche de l'étiquette dans la table des symboles
(4) impossibilité d'effectuer des RETURN multiples (CLINT et HOARE 1972), i.e. de faire communiquer des PROGS dynamiquement imbriqués.

Pour toutes ces raisons, l'utilisation de LISP sur mini-ordinateurs est incompatible avec l'usage des PROGS, la lenteur d'execution et l'encombrement que leur usage implique interdisant l'introduction et l'évaluation de programmes LISP non-triviaux dans des ressources limitées.

L'unique, et considérable intérêt des formes de type PROG réside dans la fonction RETURN, donnant la possibilité de sortir d'un bloc à partir d'un de ses points quelconques, retour accompagné d'une valeur, celle-ci se donnant alors pour la valeur de l'appel du PROG. Toutefois, la fonction RETURN ne permettait d'effectuer qu'un retour à la fonction directement appelante du PROG dans laquelle elle était dynamiquement incluse, et ne permettait pas un retour à un niveau appelant quelconque.

C'est à ses difficultés que tente de porter remède l'introduction de la fonction ESCAPE en VLISP que nous examinerons plus loin.

Voici en VLISP, l'implémentation de PROG

```
PROG_fsubr =
          (A1 NIL PLOC PETIQ : P) :: ((!X ?A1) !X ?P);
          app LIER; (P A1) :: (!PLOC !A2);
          tantque A2 :: (!X ?A2) faire
                  si X≤HATOM alors (X A2 : P) :: !P fsi
          ftan
          (P A1) :: (!PETIQ !A2);

PROG2     = tantque A2 :: (!X ?A2) faire
                  si     X≤HATOM alors
                  sinon (X A2 : P) :: (!A1 ?P);
                         rec EVAL ; P :: (!A2 ?P)
                  fsi
          ftan
          PLOC :: !P; app DELIER; P :: (!PLOC !PETIQ ?P);
          derec

RETURN_subr =
          jusqua LLIEN :: ,PLOC faire LLIEN :: !P; app DELIER fjusq
          PLOC :: !P; app DELIER; P :: (!PLOC !PETIQ ?P);
          derec
```

(1) la possibilité d'une inclusion d'une forme PROG dans le corps d'une fonction récursive rendrait très lourde à l'interprétation l'implémentation de ce balayage comme une mémo-fonction (MICHIE 1967) i.e. de n'effectuer la recherche des étiquettes qu'au premier appel d'un PROG, la table des symboles ainsi constituée restant consultable lors des appels suivants.

La fonction de type SUBR (GOTO e) permet de ne pas spécifier à l'avance l'étiquette de branchement, mais de considérer celle-ci comme le résultat de l'évaluation de la forme e. Cette fonction permet de lever l'ambiguïté inhérente au GO de MACLISP (WHITE 1970), où l'expression e est considérée comme une étiquette si e est un atome, et comme une expression à évaluer devant ramener une étiquette si e est non-atomique.

GO_{fsubr} = A1 :: (!A1); *app* GOTO

$GOTO_{subr}$ =
jusqua LLIEN :: , PLOC *faire* LLIEN :: !P; *app* DELIER *fjusq*
PETIQ ::!P; P :: !X;
jusqua X :: (,A1 !A2 ?X) *faire fjusq*
app PROG2

Notre implémentation du PROG est proche de celle de LISP 1.6, les tables d'étiquettes étant implantées, en VLISP T1600, dans la zone de contrôle, repérées par le pointeur PETIQ. PLOC est un pointeur sur le vecteur de liaison des variables locales du PROG. Ce pointeur, repérant la même sous-zone de contrôle que LLIEN, lorsque la forme évaluée est statistiquement incluse dans le corps du PROG, devient nécessaire lorsqu'un (GO !etiquette) est évalué dans le corps d'une fonction appelée par le PROG.

exemple : (DE FOO (...) ... (GO E) ...)
(PROG (...) ... (FOO ...) ... E (...) ...)

5.8 La fonction ESCAPE

La fonction ESCAPE est la fonction de saut la plus puissante compatible avec la structure de contrôle récursive de LISP. Elle n'existe avec cette généralité qu'en VLISP.

Elle se présente sous la forme

(ESCAPE f e_1 ... e_n)

où e_1 ... e_n est une suite d'expressions à évaluer, et f est un atome choisi par l'utilisateur, qui, dans la portée du bloc-ESCAPE, acquiert le statut de fonction de sortie de bloc.

ESCAPE évalue les e_1 ... e_n en séquence et

(1) si dans la portée du bloc-ESCAPE, est demandée l'évaluation de (f ob_1...ob_m) les ob_1...ob_m seront évalués en séquence, et on sort du bloc-ESCAPE en ramenant la valeur de ob_m

(2) si aucune forme de type de type (f ...) n'est évaluée, la valeur du ESCAPE sera alors celle de e_n.

Bien entendu, toutes les variables ayant pu être liées dans la portée du bloc-ESCAPE, seront restituées avec leur valeur antérieure.

exemple : (ESCAPE ASSEZ
(WHILE L (AND (LT (NEXTL L) 0) (ASSEZ L)))

Cette forme recherche dans la liste L un segment dont le CAR soit négatif. Si ce segment est obtenu, L devient un pointeur sur ce segment, dans le cas contraire la valeur de la forme est NIL.

exemple :

```
(DE COPY-IF-P (E P)
     (ESCAPE NO-P
             (COPY-IF-P-2 E)))

(DE COPY-IF-P-2 (E) (COND
     ((ATOM E) (COND
                   ((P E) E)
                   ((NO-P (LIST 'NOT E)))))
    ((CONS (COPY-IF-P-2 (NEXTL E))
           (COPY-IF-P-2 E)))))
```

La fonction COPY-IF-P livre une copie de l'expression E si tous les atomes ayant des occurences dans E satisfont à la condition P, et si ce n'est pas le cas livre une liste de la forme

(NOT atome-ne-satisfaisant-pas-la-condition)

Apparentée à l'opérateur J de LANDIN (LANDIN 1965, REYNOLDS 1972), la fonction ESCAPE permet de sortir de tout bloc-ESCAPE *quelque soit la profondeur d'imbrication dynamique* de l'appel de la fonction de sortie. Elle permet ainsi d'éviter l'usage des PROGS, tout en conservant la notion de branchement-de-retour (CLINT et HOARE 1972), en fonctionnalisant la notion d'étiquette.

Elle a de surcroît l'avantage

(1) d'être indépendante du type de bloc (WHILE, MAP, PROG, λ-expression, etc.) dans lequel la fonction de sortie est évaluée *(1)*. Ainsi les propriétés de la fonction ESCAPE seront conservées quelques soient les extensions de structures de contrôle qui pourront être incorporées à VLISP
(2) de mettre en évidence ce fait que la programmation sans branchement n'implique pas pour autant l'abandon des étiquettes (la fonction de sortie de ESCAPE *est* une étiquette)
(3) de ne pas exiger de l'utilisateur d'avoir à prévoir le nombre d'imbrications statiques ou dynamiques à franchir pour sortir du bloc-ESCAPE (ce qui est une contrainte insupportable si la langage est interprété, et n'offre pas de difficultés insurmontables si le langage est compilé)
(4) de permettre une implémentation très aisée, dans une extension de LISP telle que CONNIVER, des fonctions de sortie de type ADIEU (Mc DERMOTT 1974)
(5) d'autoriser des imbrications arbitrairement complexes de blocs-ESCAPE, chacun des blocs imbriqués pouvant appeler la fonction de sortie d'un bloc imbriquant quelconque.

Voici son implémentation

$ESCAPE_{fsubr}$ =

```
A1 :: (!F ?A1);
('ESC F F.CVAL (marq : LLIEN) PLOC PETIQ : P) :: (!F.CVAL ?P);
P :: !LLIEN; rec PROGN;
app DELIER; P :: (!PLOC !PETIQ ?P);
derec
```

ESC_{fsubr} =

```
(VARESC : P) :: !P; rec PROGN;
(P LLIEN) :: ((!VARESC ?-) !P);
jusqua P :: (,VARESC ?-) faire app DELIER fjusq
app DELIER; P :: (!PLOC !PETIQ ?P);
derec
```

(1) par opposition aux solutions proposées en BLISS (WULF 1971) dans lequel chaque structure de contrôle définit sa fonction de sortie particulière (EXITLOOP, EXITBLOCK, EXITCOND, etc.) faute d'étiquettes.

5.9 La fonction de filtrage

La notation par filtrage qui nous a permis de décrire l'implémentation de VLISP est bien entendu implémentée dans toute sa généralité en VLISP sous la forme d'une fonction standard de type SUBR :

(MATCH filtre argument)

Les caractères préfixes "!", "?" et "," indiquant le type des variables locales aux filtres sont implémentés par macro-génération (cf. §5.4).

Exemples :

La fonction PAL teste si son argument est ou non un palindrome

```
(DE PAL (L X)
   (MATCH '(*ALT* ()
                  (!-)
                  (!X ?(L (PAL ,L)) !X))
          L))
```

La fonction ISLAM teste si son argument est ou non une lambda-expression au sens du λ-calcul

```
(DE ISLAM (E X)
   (MATCH '(*ALT* !(E (ATOM ,E) (NEQ ,E 'λ))
                  (!(E (ISLAM ,E)) !(E (ISLAM ,E)))
                  (  !(X (ATOM ,X) (NEQ ,X 'λ))
                     !(E (ISLAM ,E))))
          E))
```

6 CONCLUSION

Nous avons pu présenter l'implémentation du noyau du système VLISP grâce à l'utilisation extensive de la méthode de description par filtrage. Cette présentation vise à un double but : lever une certaine confusion concernant le problème de la standardisation de LISP, et établir une méthode de spécification de LISP suffisamment précise pour pouvoir décrire adéquatement une implémentation particulière, et suffisamment générale pour pouvoir également rendre compte sans difficuité des langages dérivés de LISP dans lesquels sont introduites de nouvelles structures de données et des structures de contrôle incompatibles avec la structure récursive de LISP.
La préoccupation de standardisation de LISP vise à permettre une relative aisance dans le transport de programmes. Nous pensons que cette préoccupation est en porte à faux au regard des raisons réelles de la diversité des systèmes LISP : la plupart des utilisateurs de LISP en sont également des implémenteurs.
Ainsi ce dont nous avons besoin, ce n'est pas d'une unification de la structure externe de LISP, structure essentiellement plastique et malléable, mais tout au contraire nous avons besoin d'une standardisation de ses méthodes d'implémentation. Nous avons tenté d'en livrer une spécification précise et générale dans un langage adéquat.
La notation par filtrage a l'avantage de rendre les structures de contrôle et les structures de données LISP *visibles*, et de les rassembler dans une description unifiée. Notre unité de description, le filtrage à double continuation, est proche de la structure des *acteurs* (HEWITT 1975), toutefois le langage de spécification PLASMA qui en est issu nous paraît un langage de description d'intentions de programmes interprétés, et non pas un langage adéquat de spécification d'interprètes.
En nous restreignant volontairement au noyau de l'interprète, nous n'avons pas évoqué ici les solutions particulières que nous avons adopté, en ce qui concerne VLISP T1600, pour des problèmes tels que la gestion des différents espaces de mémoire réelle ou virtuelle. Nous n'avons pas non plus décrit les différents éléments de l'environnement naturel de cette implémentation : Prettyprint, traces, assembleur LAP, compilateur, indexeur de fonctions. Nous pensons que ces questions ne relèvent pas d'une standardisation mais appellent au contraire des solutions spécifiques à des implémentations particulières.
De même, pour permettre la description des langages dérivés de LISP, la technique du filtrage nous a permis de ne pas faire d'hypothèses trop spécifiques concernant l'implémentation de la zone de contrôle, contrairement à (BOBROW 1973). Ainsi, l'extension de la gestion de la zone de contrôle à des modes non récursifs : coroutines, backtracking, structures "à la CONNIVER", est quasi-immédiate.
Enfin, la description par filtrage nous paraît une représentation très fidèle des objets mis en jeu dans l'activité concrète du programmeur, en particulier en ce qui concerne les structures de données : un filtrage étant simultanément la représentation d'une structure de données et la spécification d'un processus.
Il n'est pas surprenant que ce soit dans le contexte de LISP que la nécessité de cette unification apparaisse avec le plus de netteté.

BIBLIOGRAPHIE

BAUMGART B.G. : *MICRO-PLANNER Alternate Reference Manual,* Stanford University Artificial Intelligence Laboratory, Operating Note n° 67, April 1972.

BERTHOD M. : *FORLISP : un LISP hyperportable,* Rapport Laboria, IRIA, Fevrier 1976.

BOBROW D.G. : *(LISP Bulletin),* ACM SIGPLAN Notices, vol.4, n°9, september 1969, pp. 17-45.

BOBROW D.G., WEGBREIT B. : *A Model and Stack Implementation of Multiple Environments,* Comm. ACM 16,10 (October 1973), pp. 591-603.

CHAILLOUX J. : *VLISP-10,* Université de Paris 8, Département d'Informatique, RT 17-76, Février 1976.

CLINT M., HOARE C.A.R. : *Program Proving : Jumps and Functions,* ACTA INFORMATICA 1, (1972), pp. 214-224.

DEUTSCH P., BERKELEY E.C. : *The LISP implementation for the PDP-1 computer,* in BERKELEY E.C., BOBROW D.G. (eds.), pp. 326-375, "The Programming Language LISP : Its Operations and Applications", the MIT Press, Cambridge, Mass., 1964.

FISCHER M.J. : *Lambda Calculus Schemata,* Proc. of the ACM Conference on Proving Assertions about Programs. (January 1972). Las Cruces, New Mexico, pp. 104-109.

GORDON M.J.C. : *Models of pure LISP,* Department of Machine Intelligence, Report n° 31, University of Edinburgh, 1973.

GORDON M.J.C. : *Towards a Semantic Theory of Dynamic Binding,* Stanford University Artificial Intelligence Laboratory, Report AIM-265, August 1975.

GREUSSAY P. : *Descriptions compactes d'interprètes implémentables : une application au langage CONNIVER,* 2ème Colloque International sur la Programmation, ed. B. ROBINET, Avril 1976, Paris, pp. 282-297.

HARALDSON A. : *LISP-details INTERLISP/360-370,* Uppsala University, 1975.

HEARN A.C. : *STANDARD LISP,* in BOBROW D.G. (ed) : LISP Bulletin, ACM SIGPLAN Notices, vol. 4, n°9 (September 1969), pp. 25-45.

HEWITT C.E., SMITH B. : *Toward a Programming Apprentice,* IEEE Transactions on Software Engineering, vol. SE-1, n°1, (March 1975), pp. 26-45

KOWALSKI R. : *Predicate Logic as Programming Language,* Department of Computational Logic, Memo n°70, University of Edinburgh, November 1973.

LANDIN P.J. : *A Correspondance between ALGOL 60 and Church's Lambda-notation*, I, II, Comm. ACM 8, 2 & 3, pp. 89-101 (Feb. 1965)
pp. 158-165 (Mar. 1965)

LINDSTROM G. : *Control Extensions in a Recursive Language*, BIT 13, (1973), pp. 50-70.

LUX A. : *Etude d'un modèle abstrait pour une machine LISP et de son implémentation*, thèse de 3ème cycle, Université de Grenoble, Mars 1975.

Mc CARTHY J. (a) : *LISP 1 Programmer's Manual*, March 1 1960, MIT, Cambridge, Mass.

Mc CARTHY J. (b) : *Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I*, Comm. ACM 3,3 (March 1960), pp. 184-195.

Mc CARTHY J. et al. : *LISP 1.5 Programmer's Manual*, the MIT Press, Cambridge, Mass, 1962.

Mc DERMOTT D.V., SUSSMAN G.J. : *The CONNIVER Reference Manual*, MIT Artificial Intelligence Memo n° 259a, January 1974.

MICHIE D. : *Memo functions : a language feature with rote learning properties*, Department of Machine Intelligence, MIP-R-29, University of Edinburgh, 1967.

MOON D.A. : *MACLISP Reference Manual*, Cambridge, MIT Project MAC, December 1975.

MOSES J. : *The Function of FUNCTION in LISP*, MIT Artificial Intelligence Laboratory, Report 199, June 1970.

NEWEY M.C. : *Formal Semantics of LISP with Applications to Program Correctness*, Stanford University Artificial Intelligence Laboratory, AIM-257, January 1975.

NORDSTROM M. et al. : *LISP FI : a FORTRAN implementation of LISP 1.5*, Uppsala University, 1971.

QUAM L.H., DIFFIE W. : *Stanford LISP 1.6 Manual*, Stanford University Artificial Intelligence Laboratory, Operating Note 28.6, 1972.

REYNOLDS J.C. : *Definitional Interpreters for Higher Order Programming Languages*, Proc. of ACM National Meeting, Boston, August 1972, pp. 717-740.

SAMET H. : *Automatically Proving the Correctness of Translations Involving Optimized Code*, Stanford University Artificial Intelligence Laboratory, Memo AIM-259, May 1975.

SANDEWALL E. : *A proposed solution to the FUNARG problem*, Uppsale University, Report n°29, November 1970.

STRACHEY C. : *A mathematical semantics which can deal with full jumps*, Séminaires IRIA : "Théorie des algorithmes, des langages et de la programmation" ed. M. NIVAT, (Mai 1973), IRIA, pp. 175-191.

TEITELMAN W. : *INTERLISP Reference Manual*, Xerox Palo Alto Research Center, (October 1974).

WHITE J.L. : *An Interim LISP User's Guide*, MIT Artificial Intelligence Laboratory, Memo N°190, March 1970.

WULF W.A. et al. : *BLISS : A language for systems programming*, Comm. ACM 14, 12 (December 1971), pp. 780-790.